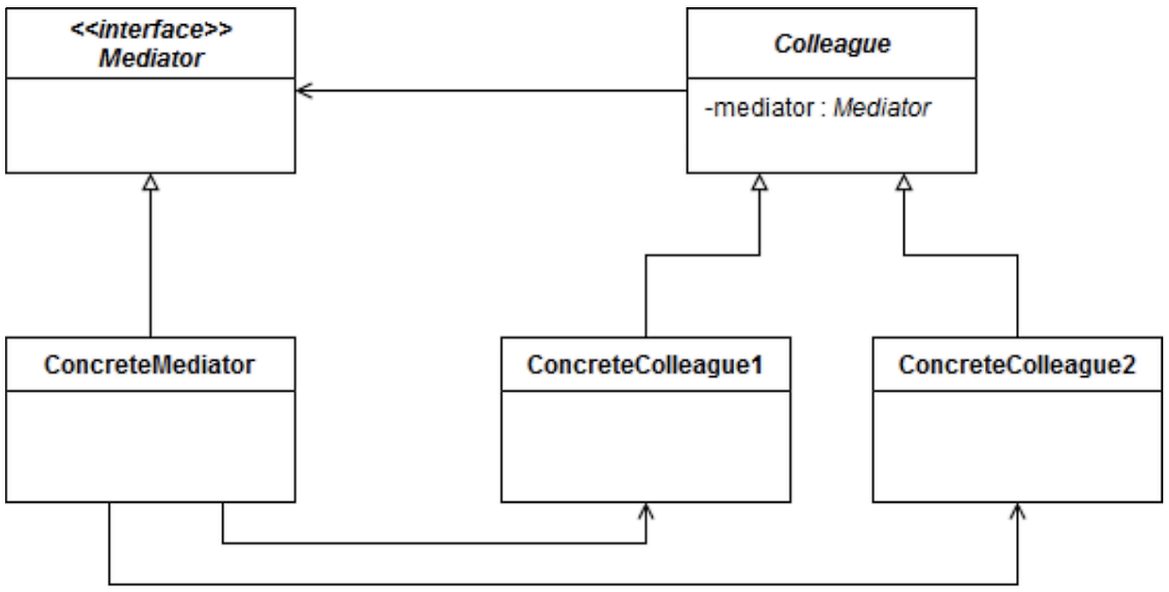
디자인 패턴

1. 행동 패턴

1) 중재자

- **중재자**는 행동 디자인 패턴이며 프로그램의 컴포넌트들이 특수 중재자 객체를 통하여 간접적으로 소통하게 함으로써 해당 컴포넌트 간의 결합도를 줄입니다. 중재자는 개별 컴포넌트들을 편집, 확장 및 재사용하는 것을 쉽게 만드는데, 그 이유는 이들이 더 이상 수십 개의 다른 클래스들에 의존하지 않기 때문입니다.
- **사용 사례들:** 자바 코드에서 중재자 패턴의 가장 인기 있는 사용 용도는 앱의 그래픽 사용자 인터페이스 컴포넌트 간의 통신을 쉽게 하는 것입니다. MVC 패턴의 컨트롤러 부분의 동의어는 중재자입니다.

중재자 패턴 구조



- Mediator - 동료 간 통신을 위한 인터페이스
- Collegue - 동료 간에 전달되는 이벤트를 정의하는 추상 클래스

- ConcreteMedediator - Colleague 객체를 조정하여 협동 조작을 구현하고 동료를 유지 관리
- ConcreteColleague - 다른 Colleague가 생성한 Mediator를 통해 받은 알림 작업을 구현

중재자 패턴 코드

1. Mediator 인터페이스

```
public interface Mediator {
        void addColleague(Colleague colleague);
        void mediate(Colleague colleague);
}
```

2. ColleagueType enum

```
public enum ColleagueType {
    USER, SYSTEM, ADMIN
}
```

3. Colleague 추상 클래스

```
public abstract class Colleague {
    private Mediator mediator;
    private String message;
    private final String name;
    private final ColleagueType type;

    protected Colleague(String name, ColleagueType type) {
        this.name = name;
        this.type = type;
    }
```

```
    public void setMediator(Mediator mediator) {
        this.mediator = mediator;
    }

    public void setMessage(String message) {
        this.message = message;
    }

    public Mediator getMediator() {
        return mediator;
    }

    public String getMessage() {
        return message;
    }

    public String getName() {
        return name;
    }

    public ColleagueType getType() {
        return type;
    }

    public void send() {
        System.out.println(this.name + " send()");
        System.out.println();
        mediator.mediate(this);
    }

    public abstract void receive(Colleague colleague);
}
```

4. Colleague 구체 클래스

```
public class UserConcreteColleague extends Colleague {
    public UserConcreteColleague(String name) {
        super(name, ColleagueType.USER);
    }

    @Override
    public void receive(Colleague colleague) {
        if (ColleagueType.SYSTEM == colleague.getType()) {
            System.out.print("[SYSTEM] ");
        } else if (ColleagueType.USER == colleague.getType()) {
            System.out.print("[" + colleague.getName() + "] ");
        }
        System.out.println(colleague.getMessage());
    }
}

public class SystemConcreteColleague extends Colleague {
    public SystemConcreteColleague(String name) {
        super(name, ColleagueType.SYSTEM);
    }

    @Override
    public void receive(Colleague colleague) {
        System.out.println("System can't receive messages");
    }
}

public class AdminConcreteColleague extends Colleague {
    public AdminConcreteColleague(String name) {
        super(name, ColleagueType.ADMIN);
    }

    @Override
    public void receive(Colleague colleague) {
        System.out.println("Admin can't receive messages");
    }
}
```

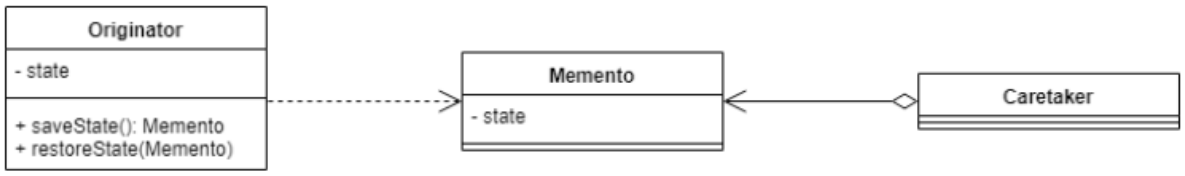
5. Mediator 테스트 코드

```
class MediatorTest {
    @Test
    @DisplayName("Mediator 테스트")
    void mediatorTest() {
        Mediator mediator = new ConcreteMediator();
        Colleague colleagueUser1 = new UserConcreteColleague("User1");
        Colleague colleagueUser2 = new UserConcreteColleague("User2");
        Colleague colleagueSystem = new SystemConcreteColleague("Syste
m");
        Colleague colleagueAdmin = new AdminConcreteColleague("Admin");

        colleagueUser1.setMediator(mediator);
        colleagueUser2.setMediator(mediator);
        colleagueSystem.setMediator(mediator);
        colleagueAdmin.setMediator(mediator);

        mediator.addColleague(colleagueUser1);
        mediator.addColleague(colleagueUser2);
        mediator.addColleague(colleagueSystem);
        mediator.addColleague(colleagueAdmin);

        colleagueUser1.setMessage("안녕하세요. User1이 보낸 메시지 입니다.");
        colleagueUser1.send();

        colleagueUser2.setMessage("안녕하세요. User2가 보낸 메시지 입니다.");
        colleagueUser2.send();

        colleagueSystem.setMessage("잠시 후 20분 뒤에 점검이 있습니다.");
        colleagueSystem.send();
    }
}
```

2) 메멘토

- **메멘토** 패턴은 행동 디자인 패턴입니다. 이 패턴은 객체 상태의 스냅숏을 만든 후 나중에 복원할 수 있도록 합니다. 메멘토는 함께 작동하는 객체의 내부 구조와 스냅숏들 내부에 보관된 데이터를 손상하지 않습니다.
- **사용 사례들:** 메멘토의 원칙은 직렬화를 사용하여 달성할 수 있으며, 이는 자바에서 매우 일반적입니다. 직렬화는 객체 상태의 스냅숏을 만드는 유일한 또는 가장 효율적인 방법은 아니나 다른 객체로부터 오리지네이터의 구조를 보호하면서 상태 백업을 저장할 수 있도록 합니다.

메멘토 패턴 구조



메멘토 패턴 코드

1. 상태 클래스 (TextWindowState)

```
public class TextWindowState {
    private String text;

    public TextWindowState(String text) {
        this.text = text;
    }

    public String getText() {
        return text;
    }
}
```

2. 원조본(Originator)

```
public class TextWindow {
    private StringBuilder currentText;

    public TextWindow() {
        this.currentText = new StringBuilder();
    }

    public String getCurrentText() {
        return currentText.toString();
    }

    public void addText(String text) {
        currentText.append(text);
    }

    public TextWindowState save() {
        return new TextWindowState(currentText.toString());
    }

    public void restore(TextWindowState save) {
        currentText = new StringBuilder(save.getText());
    }
}
```

3. Caretaker (케어테이커)

```
public class TextEditor {
    private TextWindow textWindow;
    private TextWindowState savedTextWindow;

    public TextEditor(TextWindow textWindow) {
        this.textWindow = textWindow;
    }

    public void write(String text) {
```

```
        textWindow.addText(text);
    }

    public String print() {
        return textWindow.getCurrentText();
    }

    public void hitSave() {
        savedTextWindow = textWindow.save();
    }

    public void hitUndo() {
        textWindow.restore(savedTextWindow);
    }
}
```